

HPC Toolkit: Sampling-based Performance Tools for Leadership Computing

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@cs.rice.edu



<http://hpctoolkit.org>



Acknowledgments

- **Research Staff**
 - **Nathan Tallent, Laksono Adhianto, Mike Fagan, Mark Krentel**
- **Student**
 - **Xu Liu**
- **Alumni**
 - **Gabriel Marin (ORNL), Robert Fowler (RENCI), Nathan Froyd (CodeSourcery)**
- **SciDAC project support**
 - **Center for Scalable Application Development Software**
 - Cooperative agreement number **DE-FC02-07ER25800**
 - **Performance Engineering Research Institute**
 - Cooperative agreement number **DE-FC02-06ER25762**

Challenges

- Gap between typical and peak performance is huge
- Complex architectures are harder to program effectively
 - processors that are pipelined, out of order, superscalar
 - multi-level memory hierarchy
 - multi-level parallelism: multi-core, SIMD instructions
- Complex applications present challenges
 - for measurement and analysis
 - for understanding and tuning
- Leadership computing platforms pose additional challenges
 - unique microkernel-based operating systems
 - immense scale
 - more than just computation: communication, I/O

Performance Analysis Principles

- Without accurate measurement, analysis is irrelevant
 - avoid systematic measurement error
 - instrumentation-based measurement is often problematic
 - measure actual execution of interest, not an approximation
 - fully optimized production code on the target platform
- Without effective analysis, measurement is irrelevant
 - pinpoint and explain problems in terms of source code
 - binary-level measurements, source-level insight
 - compute insightful metrics
 - “unused bandwidth” or “unused flops” rather than “cycles”
- Without scalability, a tool is irrelevant
 - large codes
 - large-scale parallelism, including MPI + OpenMP hybrid

Performance Analysis Goals

- Accurate measurement of complex parallel codes
 - large, multi-lingual programs
 - fully optimized code: loop optimization, templates, inlining
 - binary-only libraries, sometimes partially stripped
 - complex execution environments
 - dynamic loading (e.g. Linux clusters) vs. static linking (Cray XT, BG/P)
 - SPMD parallel codes with threaded node programs
 - batch jobs
- Effective performance analysis
 - insightful analysis that pinpoints and explains problems
 - correlate measurements with code (yield actionable results)
 - intuitive enough for scientists and engineers
 - detailed enough for compiler writers
- Scalable to petascale systems

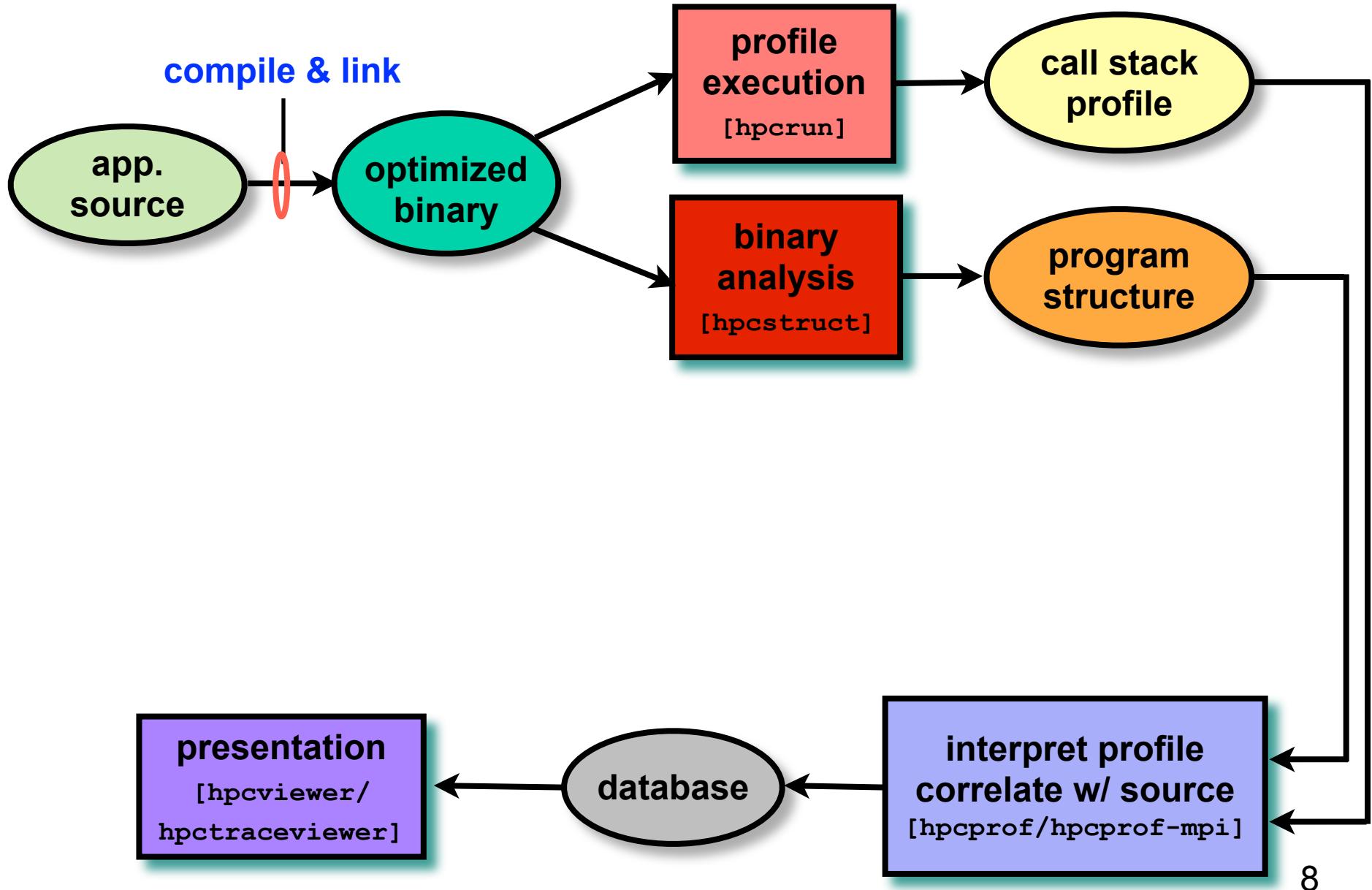
HPCToolkit Design Principles

- **Binary-level measurement and analysis**
 - observe **fully optimized, dynamically linked executions**
 - support **multi-lingual codes with external binary-only libraries**
- **Sampling-based measurement (avoid instrumentation)**
 - minimize **systematic error and avoid blind spots**
 - enable data collection for **large-scale parallelism**
- **Collect and correlate multiple derived performance metrics**
 - diagnosis requires more than one species of metric
 - derived metrics: “**unused bandwidth**” rather than “**cycles**”
- **Associate metrics with both static and dynamic context**
 - **loop nests, procedures, inlined code, calling context**
- **Support top-down performance analysis**
 - **intuitive enough for scientists and engineers to use**
 - **detailed enough to meet the needs of compiler writers**

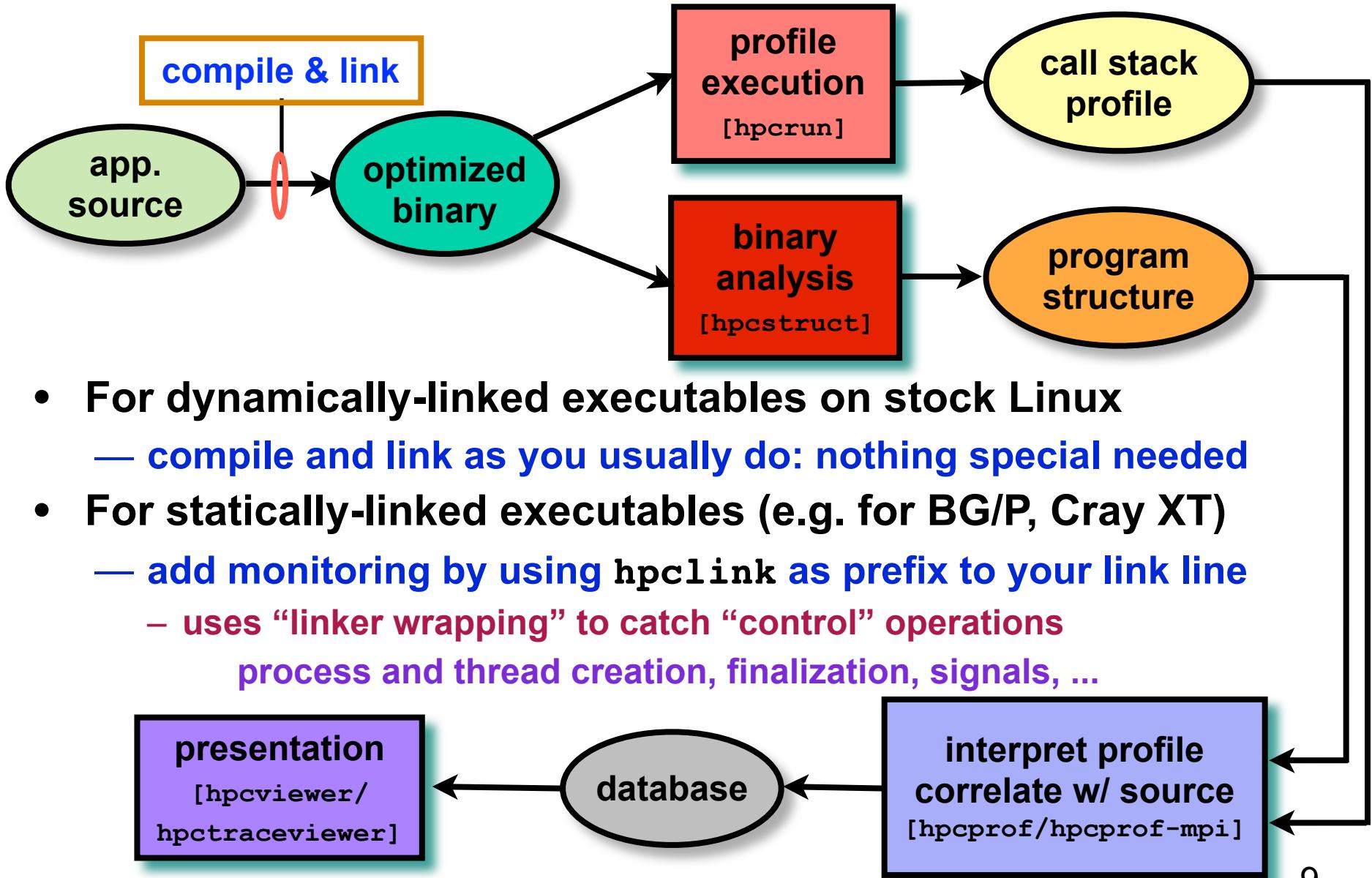
Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

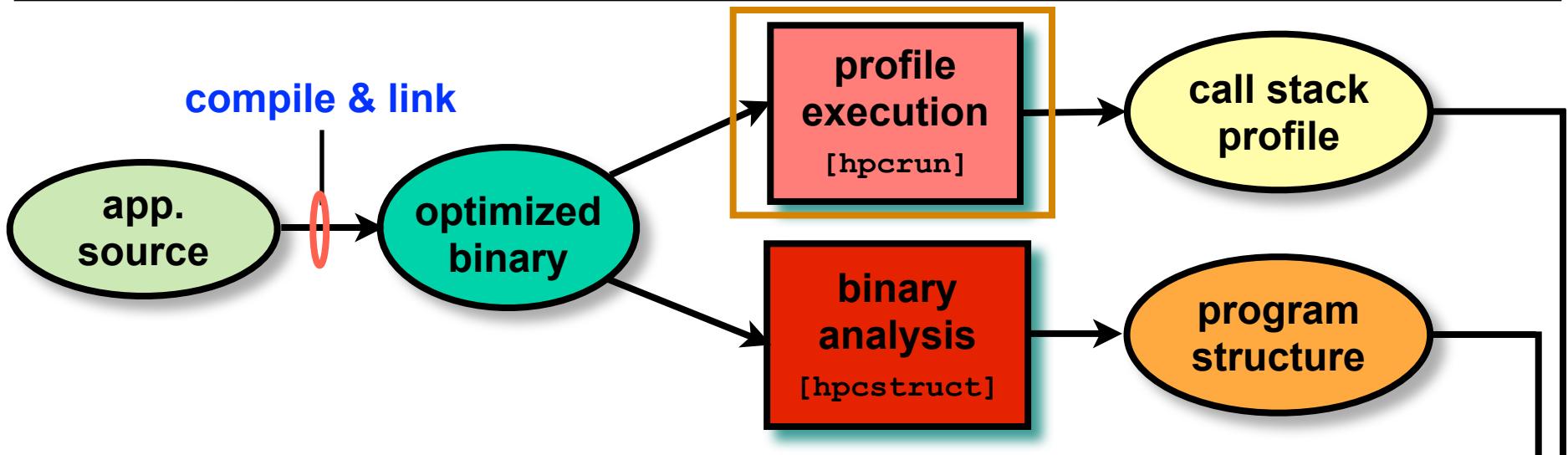
HPCToolkit Workflow



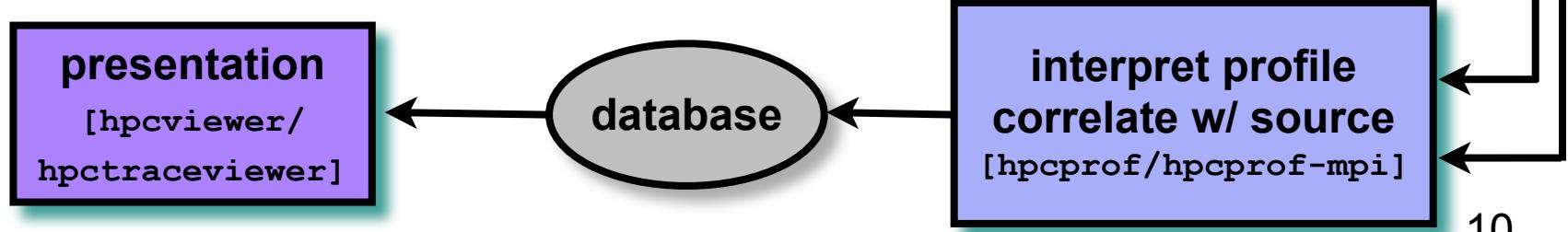
HPCToolkit Workflow



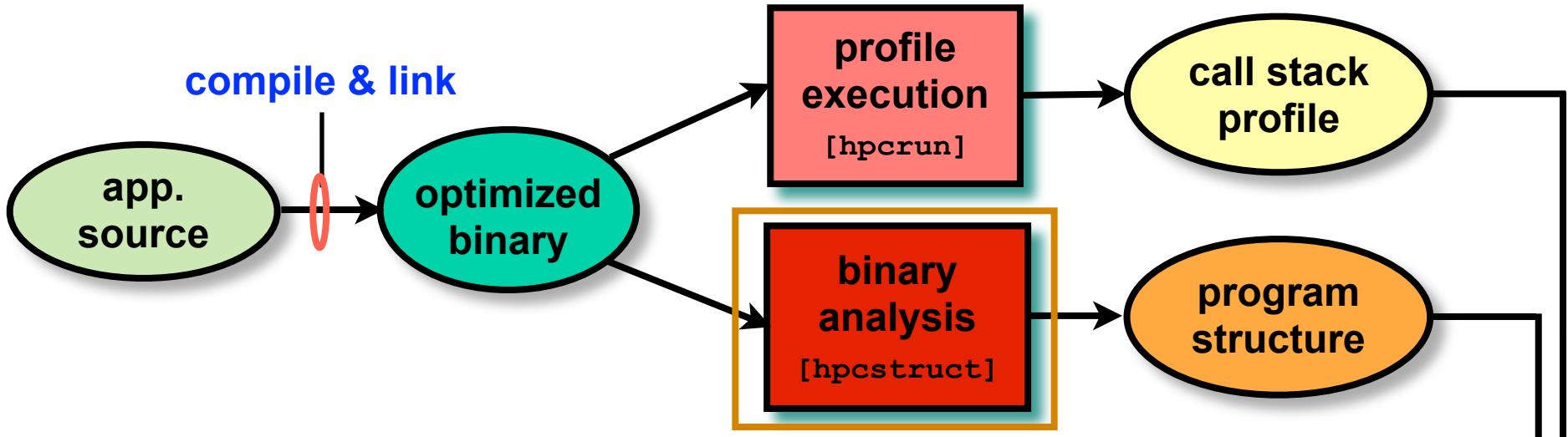
HPCToolkit Workflow



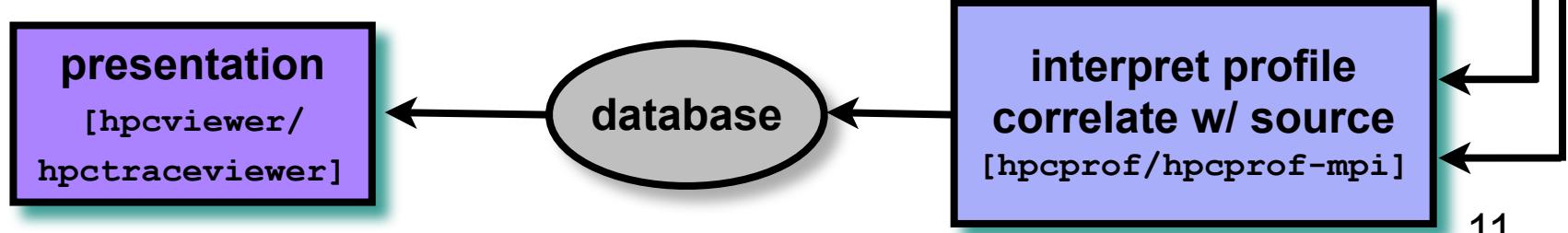
- **Measure execution unobtrusively**
 - **launch optimized application binaries**
 - dynamically-linked applications: launch with `hpcrun` to measure
 - statically-linked applications: measurement library added at link time control with environment variable settings
 - **collect statistical call path profiles of events of interest**



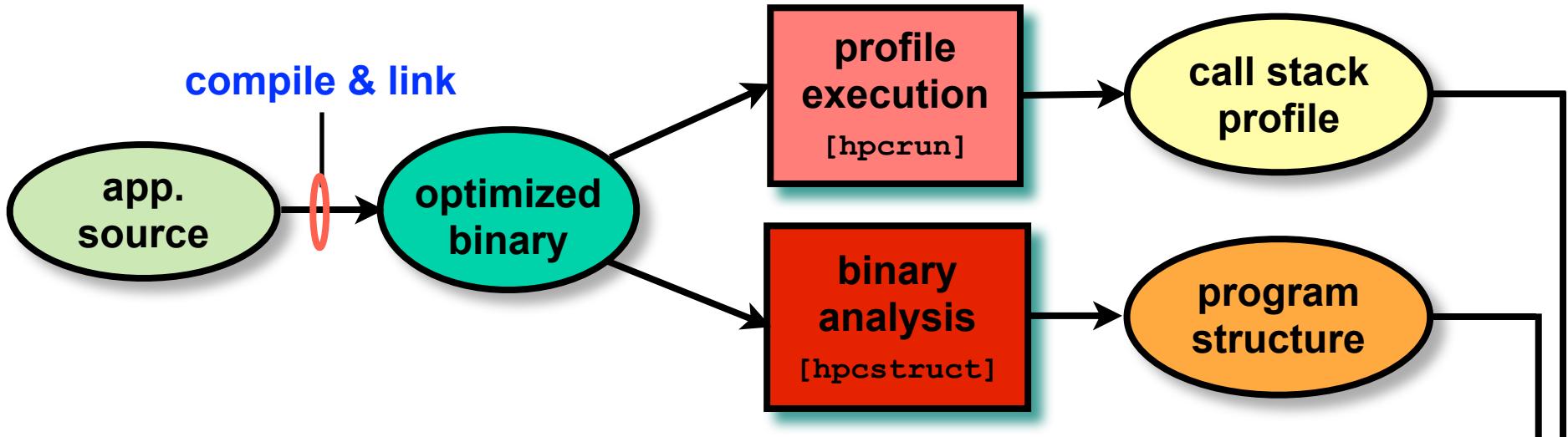
HPCToolkit Workflow



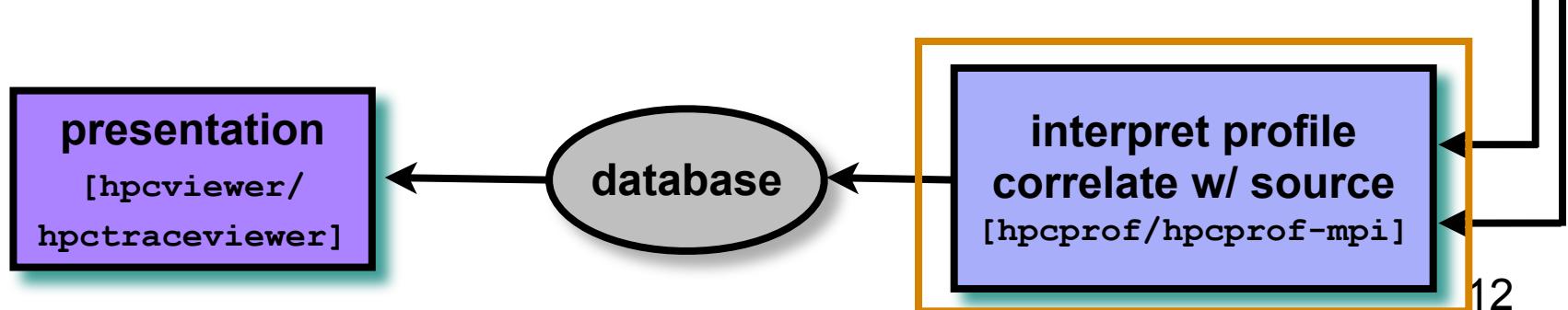
- **Analyze binary with `hpcstruct`: recover program structure**
 - analyze machine code, line map, debugging information
 - extract loop nesting & identify inlined procedures
 - map transformed loops and procedures to source



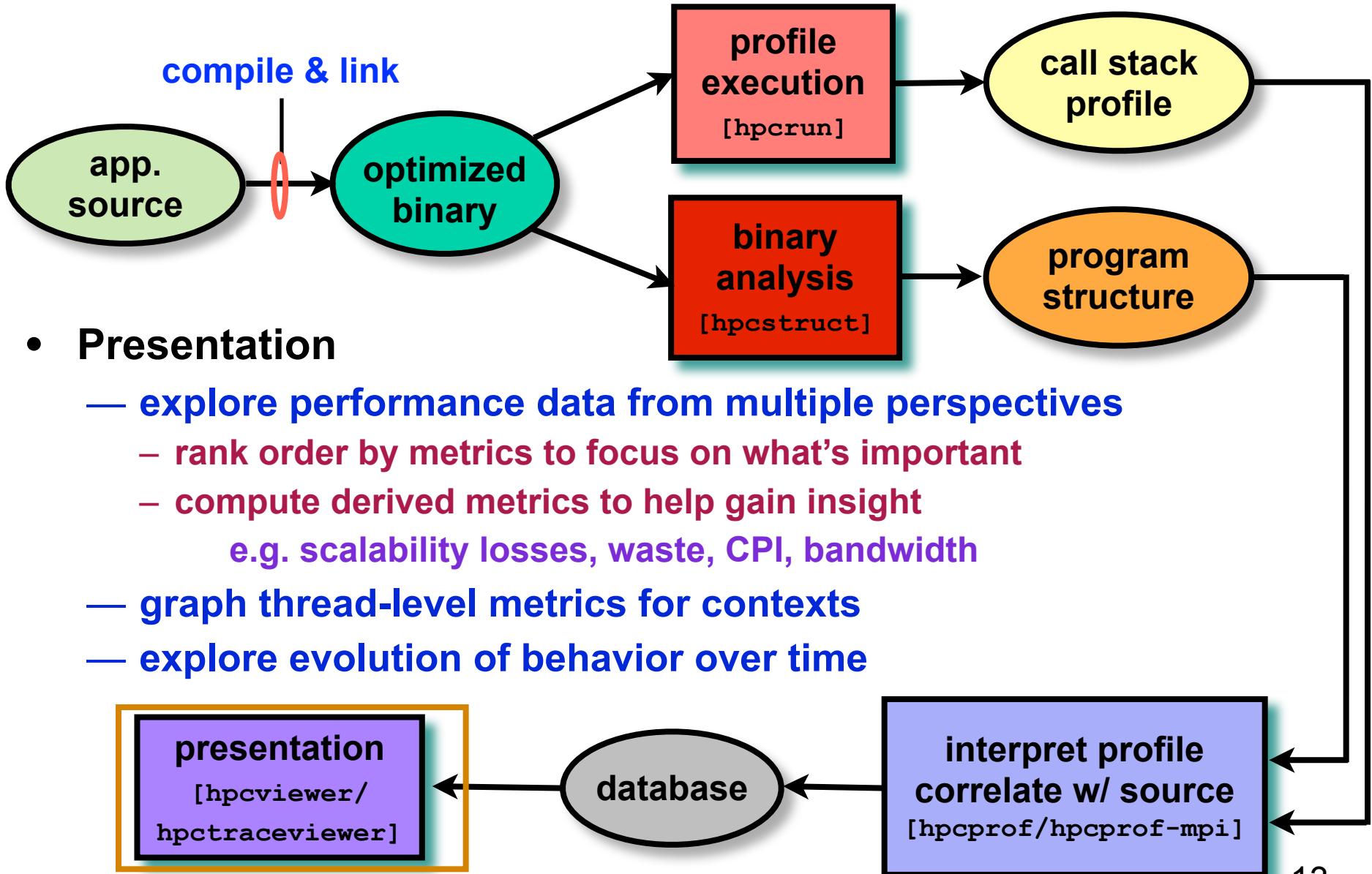
HPCToolkit Workflow



- **Combine multiple profiles**
 - **multiple threads; multiple processes; multiple executions**
- **Correlate metrics to static & dynamic program structure**



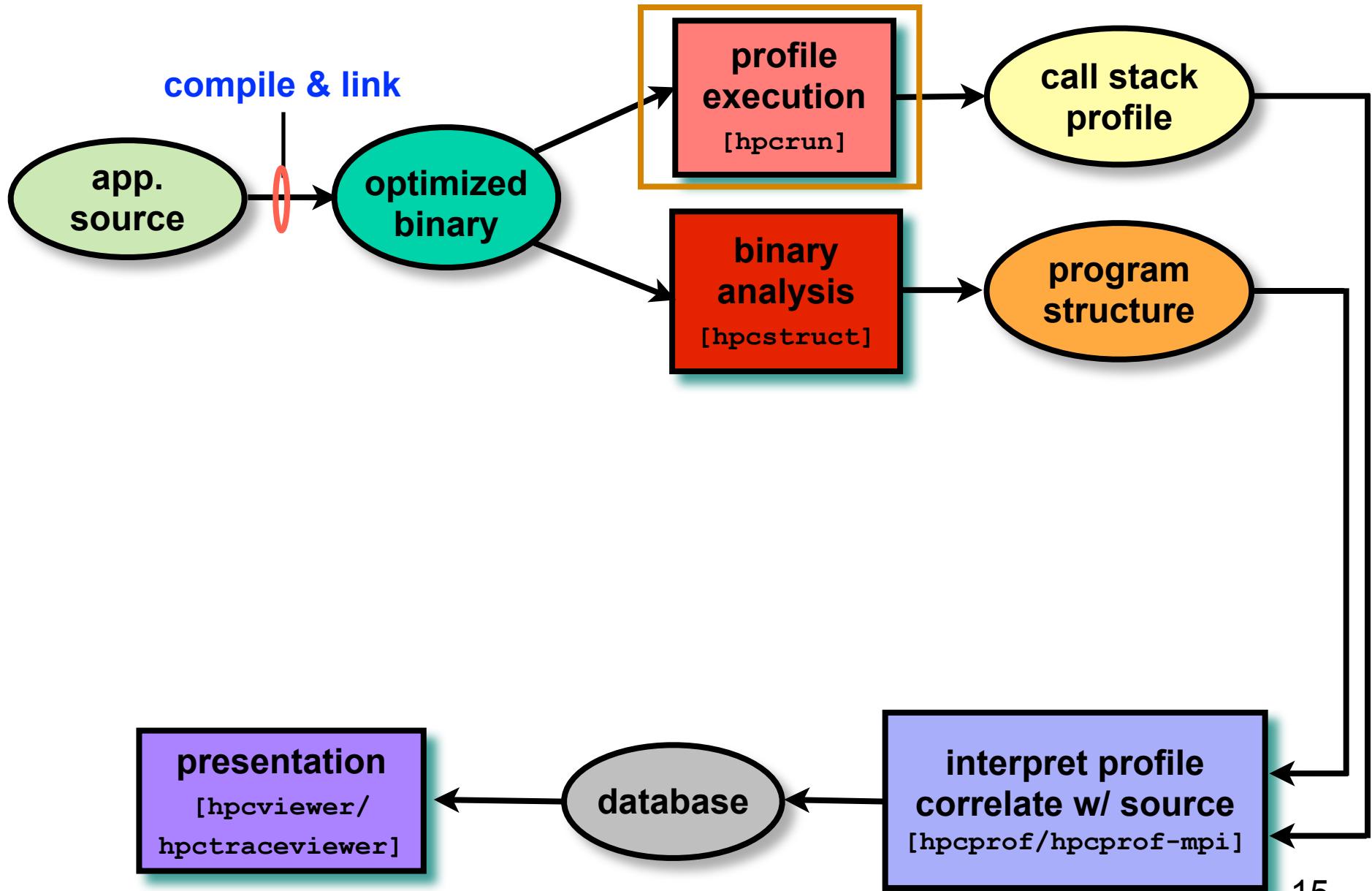
HPCToolkit Workflow



Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

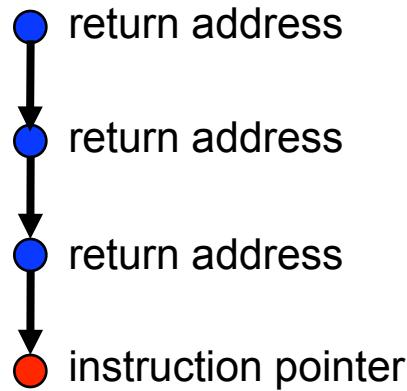
Measurement



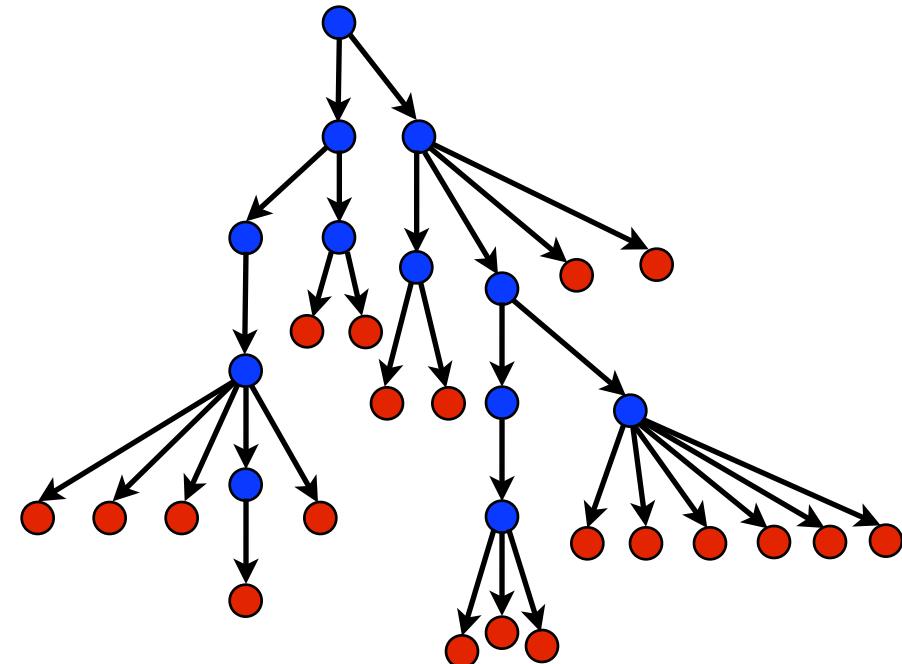
Call Path Profiling

- Measure and attribute costs in context
 - sample timer or hardware counter overflows
 - gather calling context using stack unwinding

Call path sample



Calling context tree



Overhead proportional to sampling frequency...
...not call frequency

Novel Aspects of Our Approach

- Unwind fully-optimized and even stripped code
 - use on-the-fly binary analysis to support unwinding
- Cope with dynamically-loaded shared libraries on Linux
 - note as new code becomes available in address space
- Integrate static & dynamic context information in presentation
 - dynamic call chains including procedures, inlined functions, loops, and statements

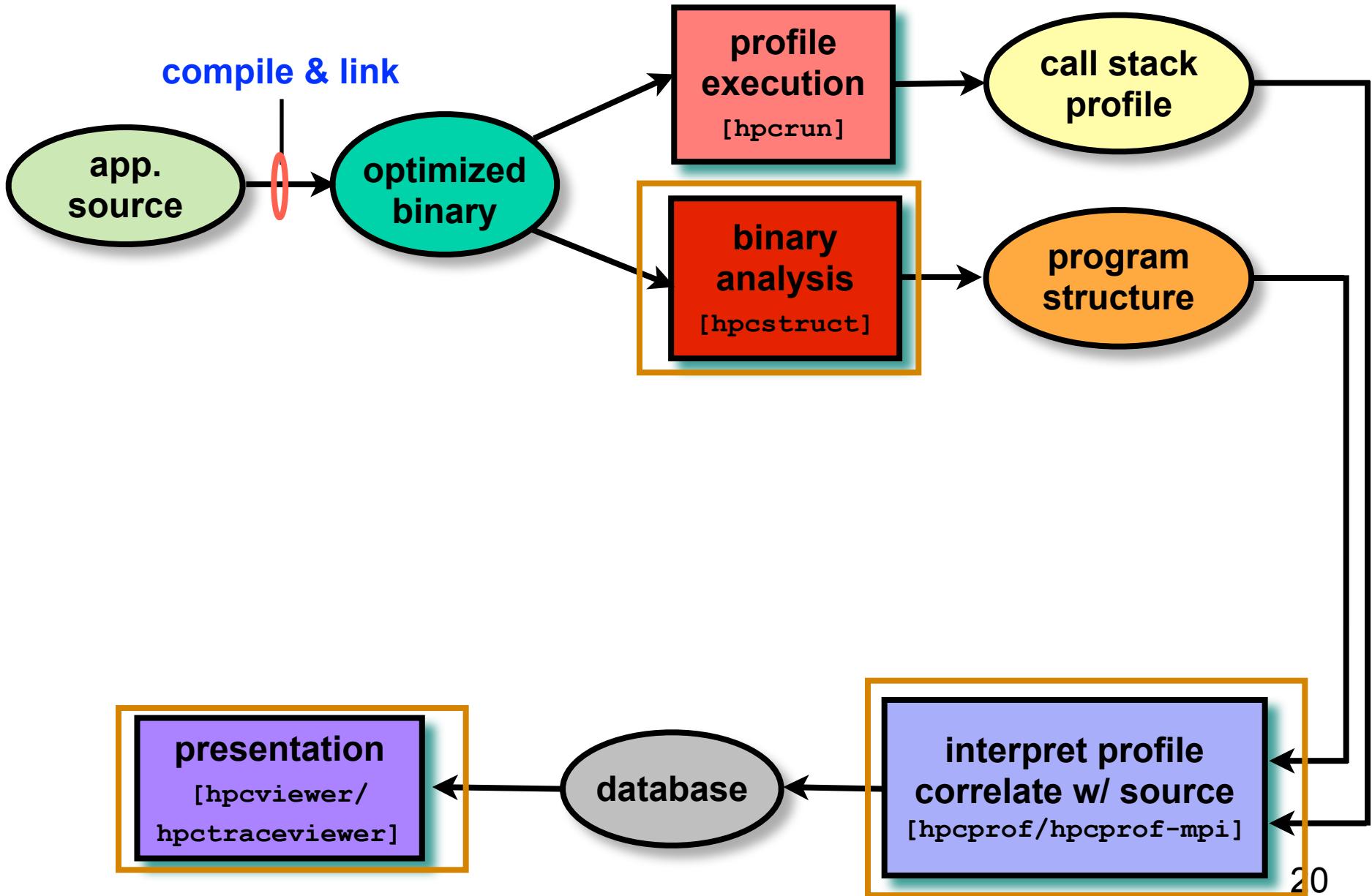
Measurement Effectiveness

- **Accurate**
 - PFLOTRAN on Cray XT @ 8192 cores
 - 148 unwind failures out of 289M unwinds
 - 5e-5% errors
 - Flash on Blue Gene/P @ 8192 cores
 - 212K unwind failures out of 1.1B unwinds
 - 2e-2% errors
 - SPEC2006 benchmark test suite (sequential codes)
 - fully-optimized executables: Intel, PGI, and Pathscale compilers
 - 292 unwind failures out of 18M unwinds (Intel Harpertown)
 - 1e-3% error
- **Low overhead**
 - e.g. PFLOTRAN scaling study on Cray XT @ 512 cores
 - measured cycles, L2 miss, FLOPs, & TLB @ 1.5% overhead
 - suitable for use on production runs

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

Effective Analysis



Recovering Program Structure

- Analyze an application binary
 - identify object code procedures and loops
 - decode machine instructions
 - construct control flow graph from branches
 - identify natural loop nests using interval analysis
 - map object code procedures/loops to source code
 - leverage line map + debugging information
 - discover inlined code
 - account for many loop and procedure transformations

Unique benefit of our binary analysis

- Bridges the gap between
 - lightweight measurement of fully optimized binaries
 - desire to correlate low-level metrics to source level abstractions

Analyzing Results with hpcviewer

hpcviewer: MOAB: mbperf_iMesh 200 B (Barcelona 2360 SE)

The screenshot shows the hpcviewer application window. At the top, there are three tabs: 'mbperf_iMesh.cpp', 'TypeSequenceManager.hpp', and 'stl_tree.h'. The 'mbperf_iMesh.cpp' tab is active, displaying C++ code for a class 'SequenceCompare'.

source pane: The code editor area where the source code is displayed.

costs for

- **inlined procedures**
- **loops**
- **function calls in full context**

view control: A panel at the bottom of the window containing buttons for 'Calling Context View', 'Callers View', 'Flat View', and other metrics like PAPI_L1_DCM and PAPI_TOT_CYC.

metric display: Buttons for navigating between different performance metrics.

navigation pane: A tree view showing the call hierarchy and specific function calls highlighted in green.

metric pane: A table showing detailed performance metrics for each function call, such as PAPI_L1_DCM and PAPI_TOT_CYC values and percentages.

	PAPI_L1_DCM (I)	PAPI_TOT_CYC (I)	P
8.63e+08 100 %	1.13e+11 100 %		
8.35e+08 96.7%	1.10e+11 97.6%		
6.81e+08 78.9%	0.98e+11 86.5%		
3.43e+08 10.9%	3.20e+10 19.3%		
3.20e+08 37.1%	2.18e+10 19.3%		
3.20e+08 37.1%	2.16e+10 19.1%		
3.20e+08 37.1%	2.16e+10 19.1%		
2.04e+08 23.7%	9.38e+09 8.3%		
2.04e+08 23.6%	9.37e+09 8.3%		
1.78e+08 20.6%	8.56e+09 7.6%		
1.78e+08 20.6%	8.56e+09 7.6%		

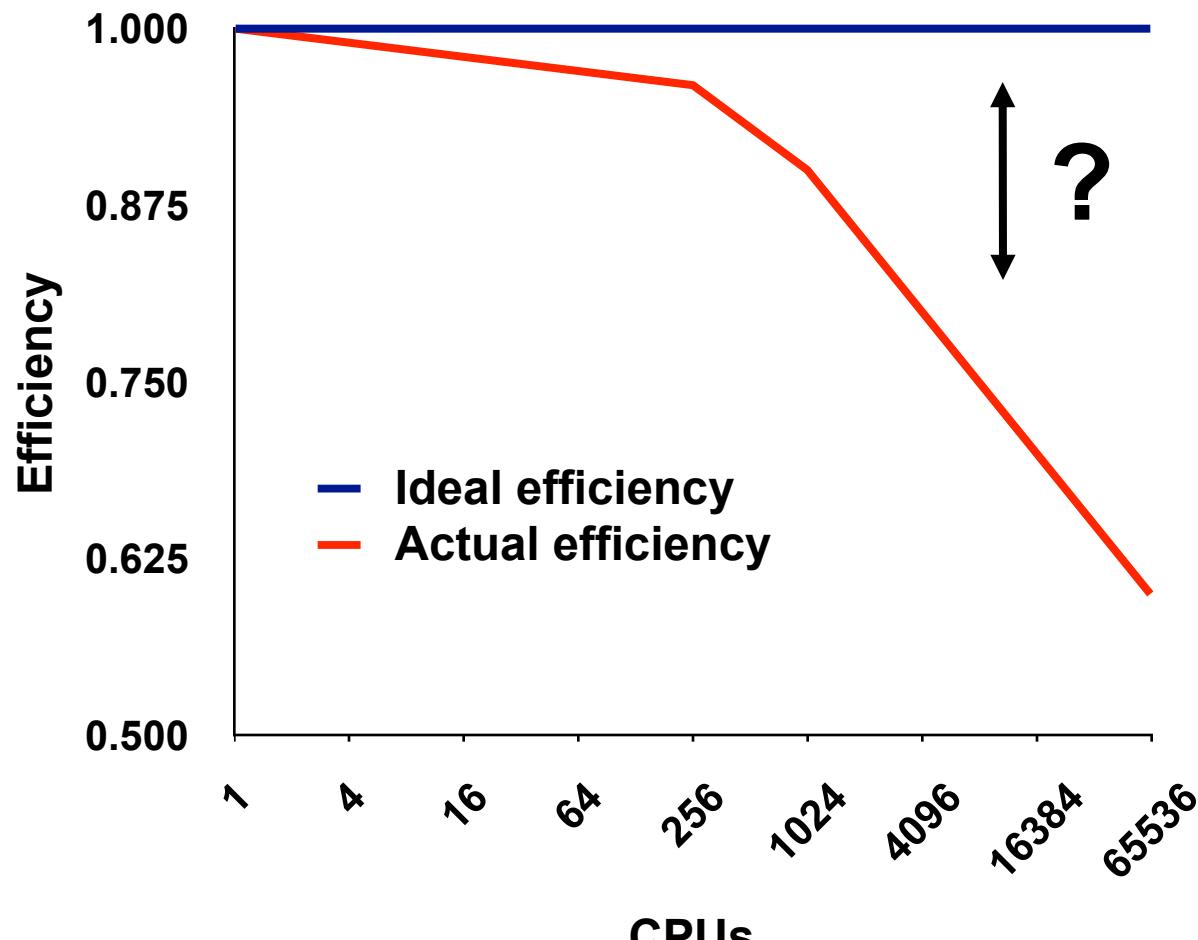
Principal Views

- **Calling context tree view** - “top-down” (down the call chain)
 - associate metrics with each dynamic calling context
 - high-level, hierarchical view of distribution of costs
- **Caller’s view** - “bottom-up” (up the call chain)
 - apportion a procedure’s metrics to its dynamic calling contexts
 - understand costs of a procedure called in many places
- **Flat view** - ignores the calling context of each sample point
 - aggregate all metrics for a procedure, from any context
 - attribute costs to loop nests and lines within a procedure

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

The Problem of Scaling



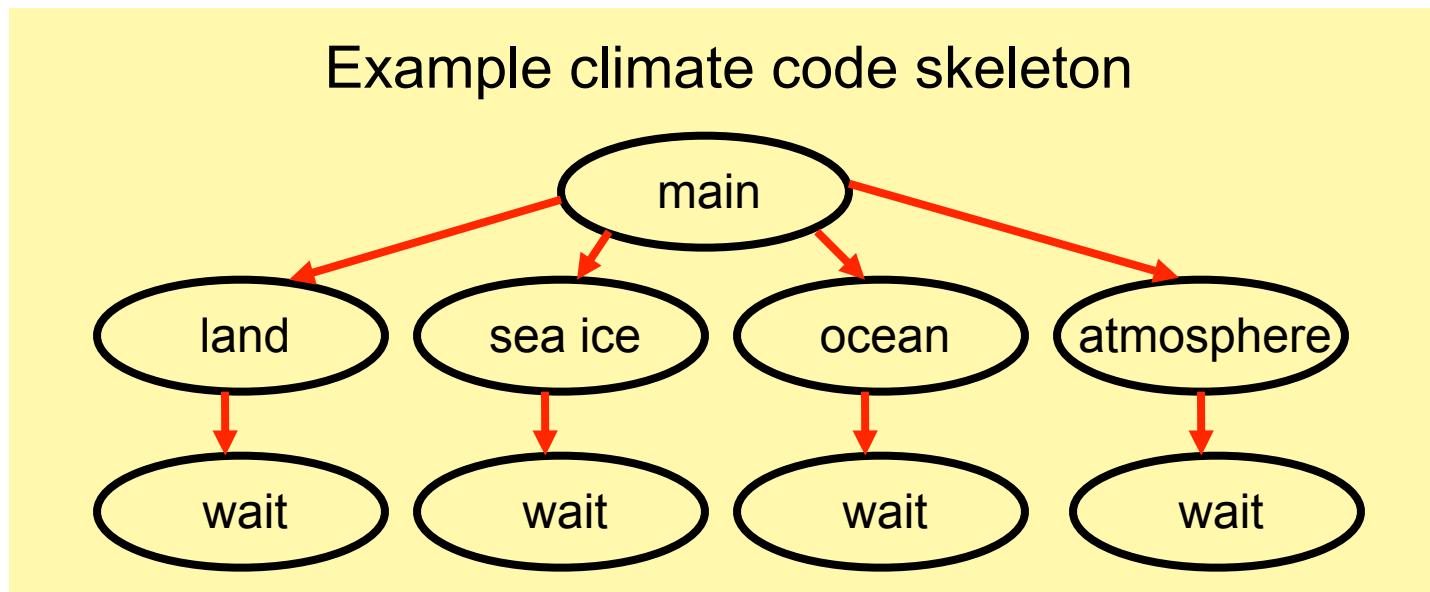
Note: higher is better

Goal: Automatic Scaling Analysis

- Pinpoint scalability bottlenecks
- Guide user to problems
- Quantify the magnitude of each problem
- Diagnose the nature of the problem

Challenges for Pinpointing Scalability Bottlenecks

- Parallel applications
 - modern software uses layers of libraries
 - performance is often context dependent
- Monitoring
 - bottleneck nature: computation, data movement, synchronization?
 - 2 pragmatic constraints
 - acceptable data volume
 - low perturbation for use in production runs



Performance Analysis with Expectations

- You have performance expectations for your parallel code
 - strong scaling: linear speedup
 - weak scaling: constant execution time
- Putting your expectations to work
 - measure performance under different conditions
 - e.g. different levels of parallelism or different inputs
 - express your expectations as an equation
 - compute the deviation from expectations for each calling context
 - for both inclusive and exclusive costs
 - correlate the metrics with the source code
 - explore the annotated call tree interactively

Weak Scaling Analysis for SPMD Codes

Performance expectation for weak scaling

- work increases linearly with # processors
- execution time is same as that on a single processor

- Execute code on p and q processors; without loss of generality, $p < q$
- Let T_i = total execution time on i processors
- For corresponding nodes n_q and n_p
 - let $C(n_q)$ and $C(n_p)$ be the costs of nodes n_q and n_p
- Expectation: $C(n_q) = C(n_p)$
- Fraction of excess work: $X_w(n_q) = \frac{C(n_q) - C(n_p)}{T_q}$ parallel overheadtotal time

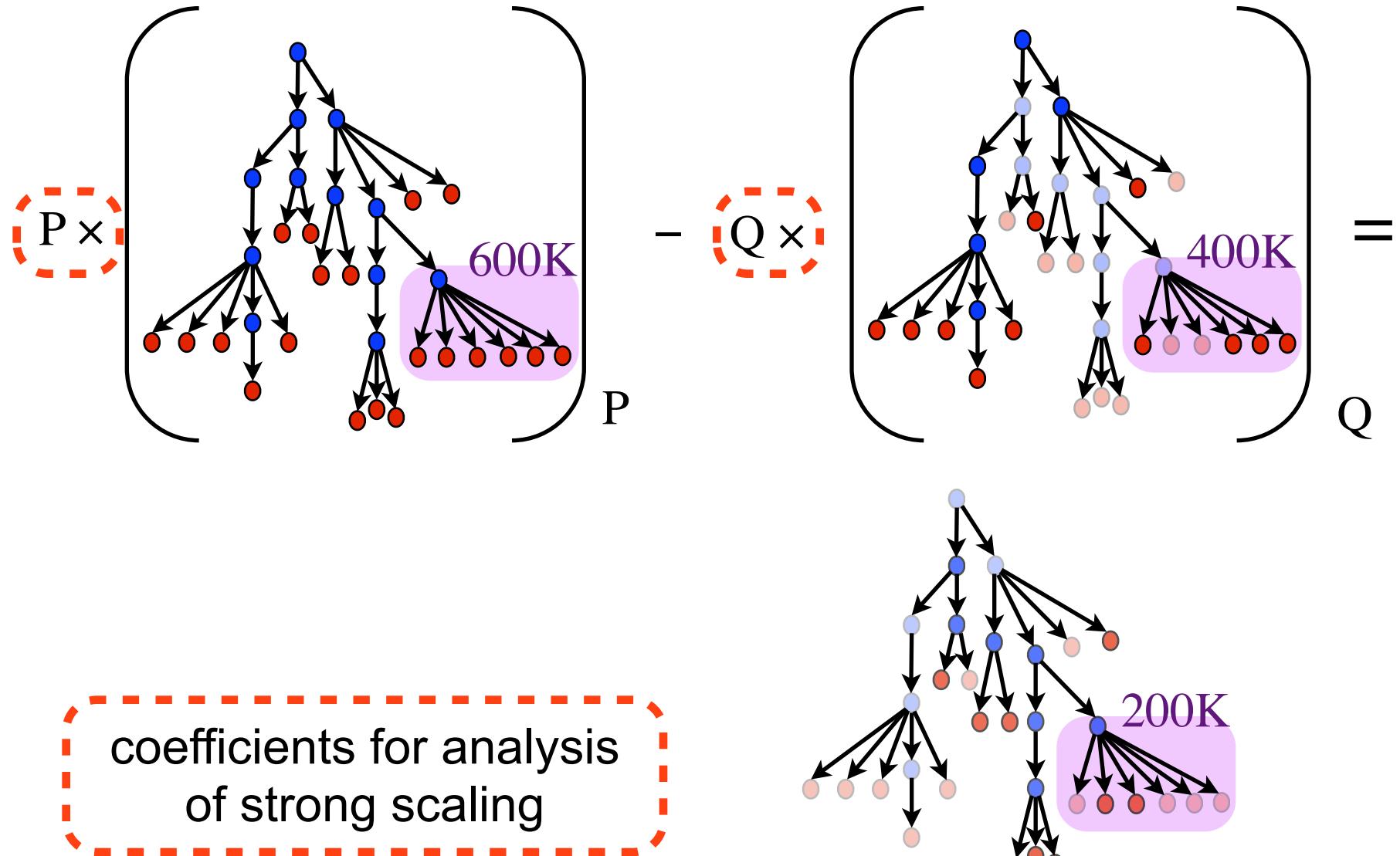
Strong Scaling Analysis for SPMD Codes

Performance expectation for strong scaling

- work is constant
- execution time decreases linearly with # processors

- Execute code on p and q processors; without loss of generality, $p < q$
- Let T_i = total execution time on i processors
- For corresponding nodes n_q and n_p
 - let $C(n_q)$ and $C(n_p)$ be the costs of nodes n_q and n_p
- Expectation: $qC_q(n_q) = pC_p(n_p)$
- Fraction of excess work: $X_s(C, n_q) = \frac{qC_q(n_q) - pC_p(n_p)}{qT_q}$ parallel overhead
total time

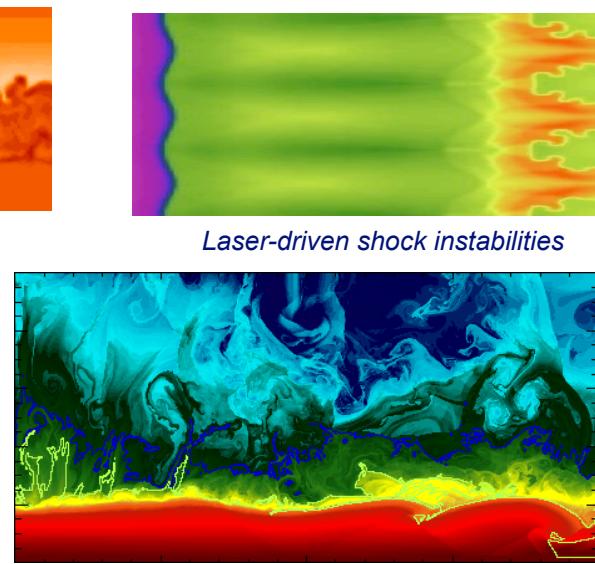
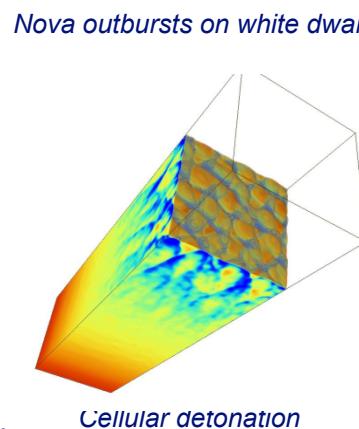
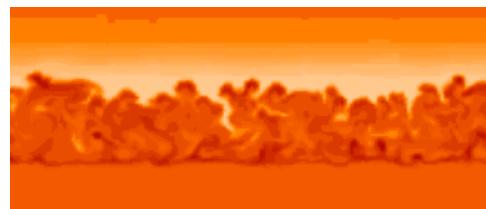
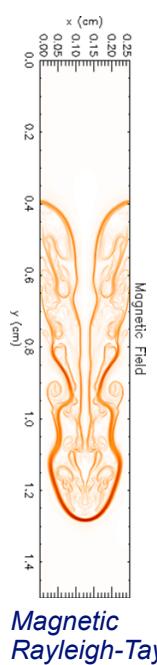
Pinpointing and Quantifying Scalability Bottlenecks



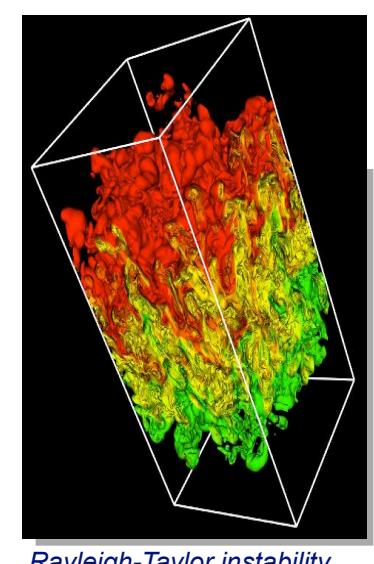
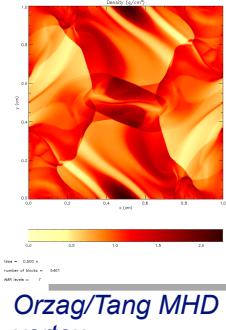
Scalability Analysis Demo

Code:
Simulation:
Platform:
Experiment:
Scaling type:

University of Chicago FLASH
white dwarf detonation
Blue Gene/P
8192 vs. 256 processors
weak



Helium burning on neutron stars



Figures courtesy of FLASH Team, University of Chicago

Scaling on Multicore Processors

- Compare performance
 - single vs. multiple processes on a multicore system
- Strategy
 - differential performance analysis
 - subtract the calling context trees as before, unit coefficient for each

S3D: Multicore Losses at the Loop Level

hpcviewer: [Profile Name]

getrates.f rhsf.f90 diffflux_gen_uj.f

```
193 *ge. 2) then
194     l_ujUpper30 = (3 - 1 + 1) / 3 * 3 + 1 - 1
195     do m = 1, l_ujUpper30, 3
196         do n = 1, n_spec - 1
197             do lt_2 = 1, nz
198                 do lt_1 = 1, ny
199                     do lt_0 = 1, nx
200                         diffflux(lt_0, lt_1, lt_2, n, m) = -ds_mixav_
201                         *(lt_0, lt_1, lt_2, n) * (grad_ys(lt_0, lt_1, lt_2, n, m) +
202                         *s(lt_0, lt_1, lt_2, n) * grad_mixmw(lt_0, lt_1, lt_2, m))
203                         diffflux(lt_0, lt_1, lt_2, n_spec, m) = diff_
204                         *lux(lt_0, lt_1, lt_2, n_spec, m) - diffflux(lt_0, lt_1, lt_2,
205                         *, n, m)
206                         diffflux(lt_0, lt_1, lt_2, n, m + 1) = -ds_m_
207                         *xavg(lt_0, lt_1, lt_2, n) * (grad_ys(lt_0, lt_1, lt_2, n, m
208                         *+ 1) + vs(lt_0 1+ 1 1+ 2 n) * grad_mixmw(lt_0 1+ 1 1+ 2
```

Calling Context View Callers View Flat View

Scope	1-core (ms) (I)	1-core (ms) (E)	8-core(1) (ms) (I)	8-core(1) (ms) (E)	Multicore Loss
loop at diffflux_gen_uj.f: 197-222	2.86e06 2.6%	2.86e06 2.6%	8.12e06 4.3%	8.12e06 4.3%	5.27e06 6.9%
loop at integrate_erk_jstage_lt_ge	1.09e08 98.1%	1.25e06 1.1%	1.84e08 97.9%	5.94e06 3.2%	4.70e06 6.1%
loop at variables_m.f90: 88-99	1.49e06 1.3%	1.49e06 1.3%	6.08e06 3.2%	6.08e06 3.2%	4.60e06 6.0%
loop at rhsf.f90: 516-536	2.70e06 2.4%	1.31e06 1.2%	6.49e06 3.5%	3.72e06 2.0%	2.41e06 3.1%
loop at rhsf.f90: 538-544	3.35e06 3.0%	1.45e06 1.3%	7.06e06 3.8%	3.82e06 2.0%	2.36e06 3.1%
loop at rhsf.f90: 546-552	2.56e06 2.3%	1.47e06 1.3%	5.86e06 3.1%	3.42e06 1.8%	1.96e06 2.6%
loop at thermchem_m.f90: 127-1	8.00e05 0.7%	8.00e05 0.7%	2.28e06 1.2%	2.28e06 1.2%	1.48e06 1.9%
loop at heatflux_lt_gen.f: 5-132	1.46e06 1.3%	1.46e06 1.3%	2.88e06 1.5%	2.88e06 1.5%	1.41e06 1.8%
loop at rhsf.f90: 576	6.65e05 0.6%	6.65e05 0.6%	1.87e06 1.0%	1.87e06 1.0%	1.20e06 1.6%
loop at getrates.f: 504-505	8.00e06 7.2%	8.00e06 7.2%	8.74e06 4.7%	8.74e06 4.7%	7.35e05 1.0%
loop at derivative_x.f90: 213-690	1.78e06 1.6%	1.78e06 1.6%	2.47e06 1.3%	2.47e06 1.3%	6.95e05 0.9%

Execution time increases 2.8x in the loop that scales worst

loop contributes a 6.9% scaling loss to whole execution

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

PFLOTRAN

8K cores, Cray XT5

1. Drill down 'hot path' to loop (a balance point)

2. Notice top two call sites...

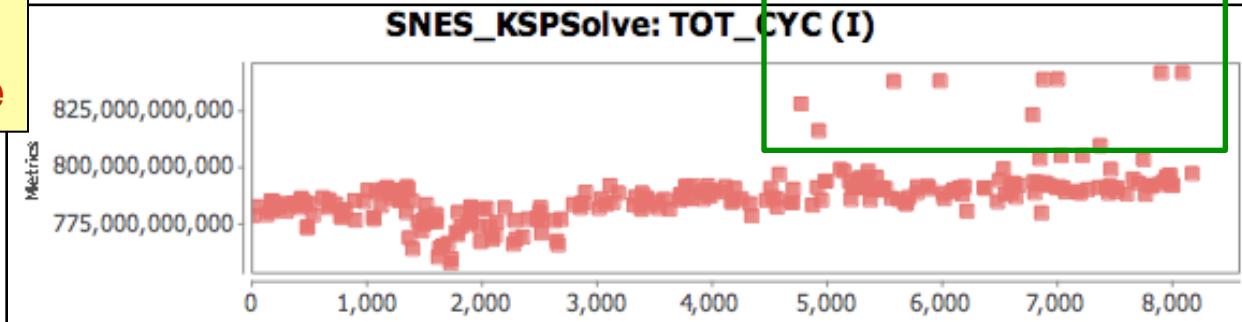
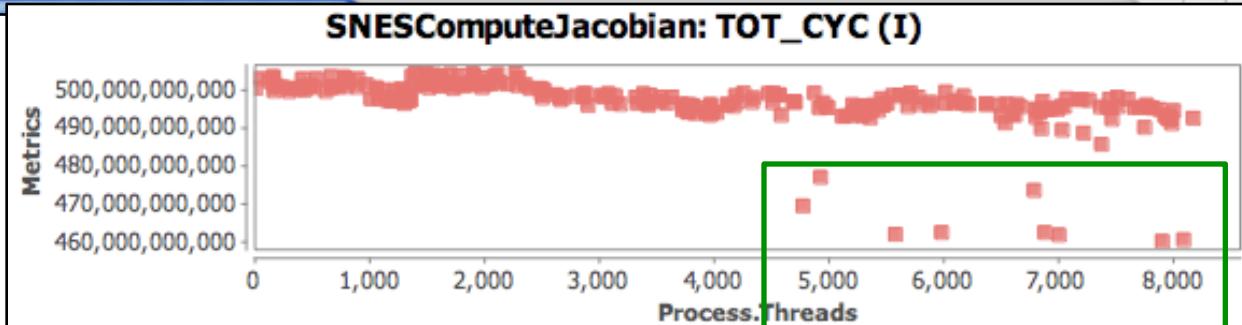
3. Plot the per-process values:

Early finishers...

... become early arrivers at Allreduce

node	imbalance (I) ▼	TOT_CYC:Sum (I)
pflotran	5.28e+15	1.85e+16 100 %
timestepper_module_stepperrun_	5.17e+15	1.82e+16 98.3%
loop at timestepper.F90: 384	5.17e+15	1.82e+16 98.2%
timestepper_module_steersteptransportdt_	2.22e+15	1.33e+16 72.0%
loop at timestepper.F90: 1230	2.22e+15	1.33e+16 72.0%
loop at timestepper.F90: 1254	2.22e+15	1.32e+16 71.3%
snessolve_	2.22e+15	1.30e+16 70.4%
SNESolve	2.22e+15	1.30e+16 70.4%
SNESSolve_LS	2.22e+15	1.30e+16 70.4%
loop at ls.c: 181	2.15e+15	1.27e+16 68.8%
SNES_KSPSolve	1.19e+15	6.44e+15 34.8%
SNESComputeJacob	6.21e+14	4.07e+15 22.0%

```
89     ierr = SNESCompute Jacobian(snes,X,&snes->jacobian,&snes->jacobian_pre,&
190     ierr = KSPSetOperators(snes->ksp,snes->jacobian,snes->jacobian_pre,flg)
191     ierr = SNES_KSPSolve(snes,snes->ksp,F,Y);CHKERRQ(ierr);
```

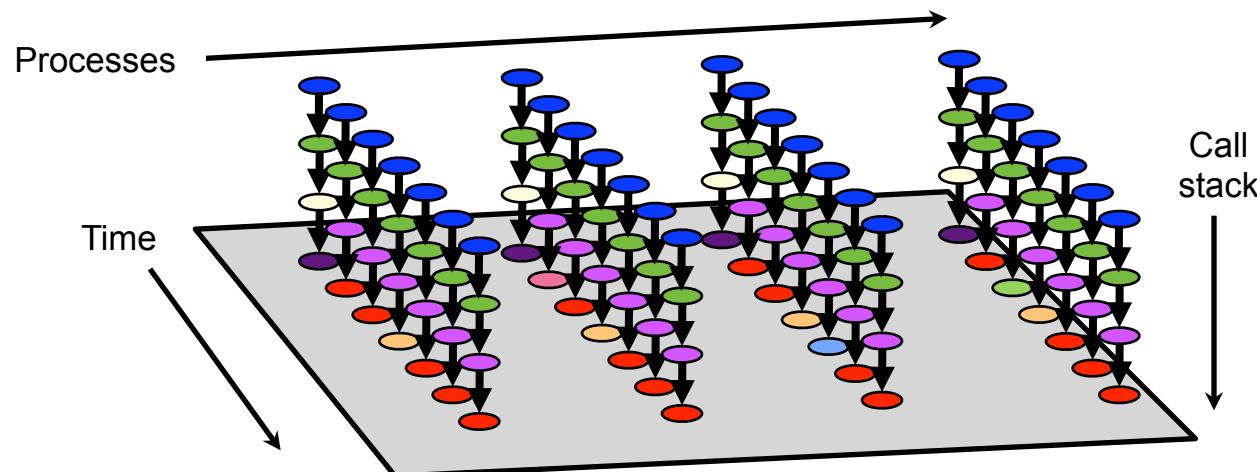


Outline

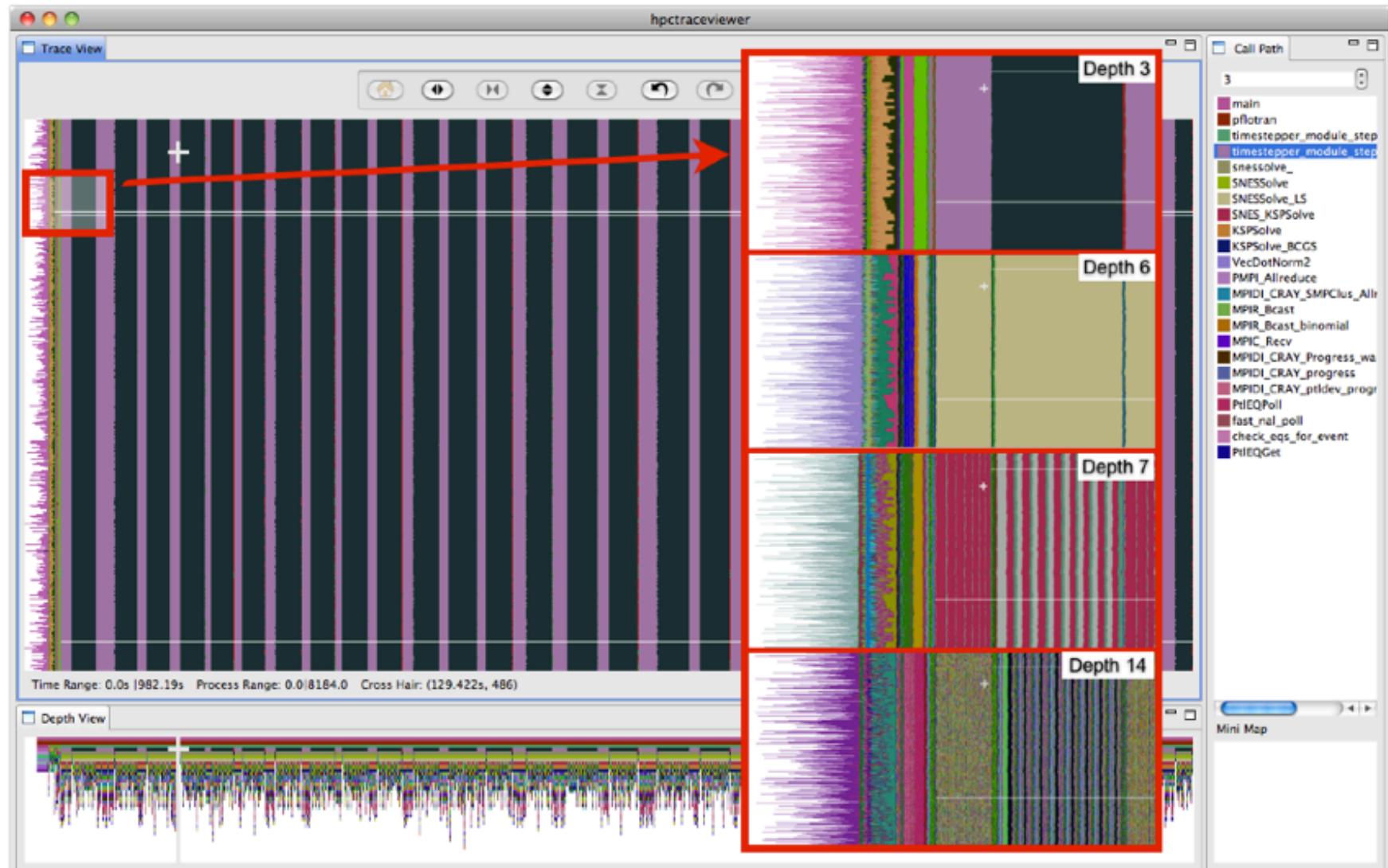
- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

Understanding Temporal Behavior

- Profiling compresses out the temporal dimension
 - temporal patterns, e.g. serialization, are invisible in profiles
- What can we do? Trace call path samples
 - sketch:
 - N times per second, take a call path sample of each thread
 - organize the samples for each thread along a time line
 - view how the execution evolves left to right
 - what do we view?
 - assign each procedure a color; view a depth slice of an execution



Process-Time Views of PFLOTRAN



8184-core execution on Cray XT5. Trace view rendered using `hpctraceviewer` on a Mac Book Pro Laptop.
Insets show zoomed view of marked region at different call stack depths.

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

Where to Find HPCToolkit

- **DOE Systems**
 - **jaguar**: /ccs/proj/hpctoolkit/pkgs/hpctoolkit
 - **intrepid**: /home/projects/hpctoolkit/pkgs/hpctoolkit
 - **franklin**: /project/projectdirs/hpctk/pkgs/hpctoolkit
- **NSF Systems**
 - **ranger**: /scratch/projects/hpctoolkit/pkgs/hpctoolkit
- **For your local Linux systems, you can download and install it**
 - **documentation, build instructions, and software**
 - see <http://hpctoolkit.org> for instructions
 - **we recommend downloading and building from svn**
 - **important notes:**
 - **using hardware counters requires downloading and installing PAPI**
 - **kernel support for hardware counters**
 - on **Linux 2.6.32 or better**: built-in kernel support for counters
 - requires PAPI newer than 4.1.1 (CVS version at present)

[earlier Linux needs a kernel patch \(perfmon2 or perfctr\)](#)

HPCToolkit Documentation

<http://hpctoolkit.org/documentation.html>

- **Comprehensive user manual:**

<http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf>

- **Quick start guide**
 - essential overview that almost fits on one page
- **Using HPCToolkit with statically linked programs**
 - a guide for using hpctoolkit on BG/P and Cray XT
- **The hpcviewer user interface**
- **Effective strategies for analyzing program performance with HPCToolkit**
 - analyzing scalability, waste, multicore performance ...
- **HPCToolkit and MPI**
- **HPCToolkit Troubleshooting**
 - why don't I have any source code in the viewer?
 - hpcviewer isn't working well over the network ... what can I do?

- **Installation guide**

Using HPCToolkit

- Add hpctoolkit's bin directory to your path
 - see earlier slide for HPCToolkit's HOME directory on your system
- Adjust your compiler flags (if you want full attribution to src)
 - add -g flag after any optimization flags
- Add hpclink as a prefix to your Makefile's link line
 - e.g. `hpclink mpixlf -o myapp foo.o ... lib.a -lm ...`
- Decide what hardware counters to monitor
 - statically-linked executables (e.g., Cray XT, BG/P)
 - use hpclink to link your executable
 - launch executable with environment var `HPCRUN_EVENT_LIST=LIST` (BG/P hardware counters supported)
 - dynamically-linked executables (e.g., Linux)
 - use hpcrun -L to learn about counters available for profiling
 - use papi_avail
 - you can sample any event listed as “profilable”

HPCToolkit Examples on Intrepid

- Example script for monitoring an application using hpctoolkit
 - </home/projects/hpctoolkit/pkgs/hpctoolkit/share/examples/bgp-scripts/run-bgp.sh>
- Example script for launching hpcprof-mpi
 - </home/projects/hpctoolkit/pkgs/hpctoolkit/share/examples/bgp-scripts/run-hpcprof-bgp.sh>
- Example performance data
 - </home/projects/hpctoolkit/pkgs/hpctoolkit/share/examples/data/hpctoolkit-fft-crayxt-256>

Launching your Job

- **Modify your run script to enable monitoring**
 - **Cray XT: set environment variable in your PBS script**
 - e.g. `setenv HPCRUN_EVENT_LIST "PAPI_TOT_CYC@3000000
PAPI_L2_MISS@400000 PAPI_TLB_MISS@400000
PAPI_FP_OPS@400000"`
 - **Blue Gene/P: pass environment settings to cqsub**
 - `cqsub -p YourAllocation -q prod-devel -t 30 -n 2048 -c 8192 \
--mode vn --env HPCRUN_EVENT_LIST=WALLCLOCK@1000 \
flash3.hpc`

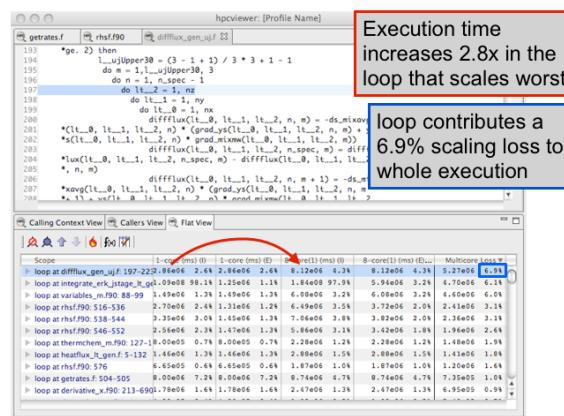
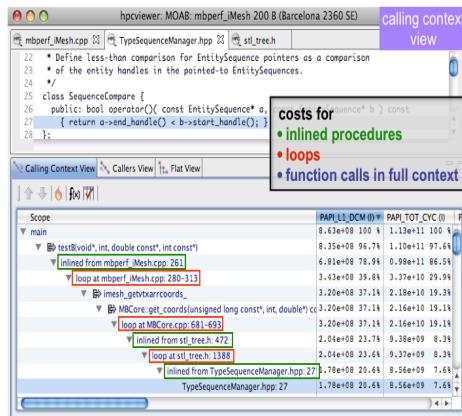
Analysis and Visualization

- Use **hpcstruct** to reconstruct program structure
 - e.g. `hpcstruct myapp`
 - creates `myapp.hpcstruct`
- Use **hpcsummary** script to summarize measurement data
 - e.g. `hpcsummary hpctoolkit-myapp-measurements-5912`
- Use **hpcprof** to correlate measurements to source code
 - run **hpcprof** on the front-end node
 - run **hpcprof-mpi** on the back-end nodes to analyze data in parallel
- Use **hpcviewer** to open resulting database
- Use **hpctraceviewer** to explore traces (collected with `-t` option)

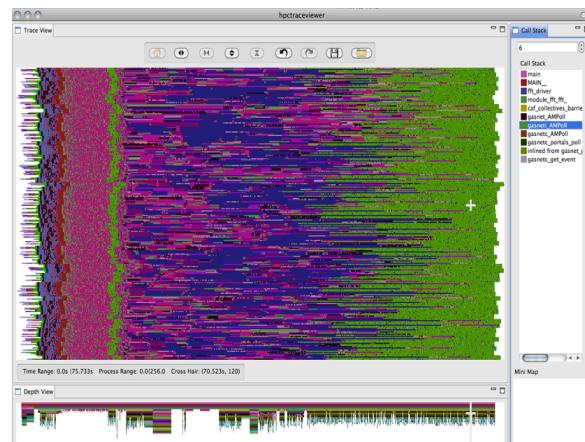
A Special Note About hpcstruct and xlf

- IBM's xlf compiler emits machine code for Fortran that have an unusual mapping back to source
- To compensate, hpcstruct needs a special option
 - **--loop-fwd-subst=no**
 - without this option, many nested loops will be missing in hpcstruct's output and (as a result) hpcviewer

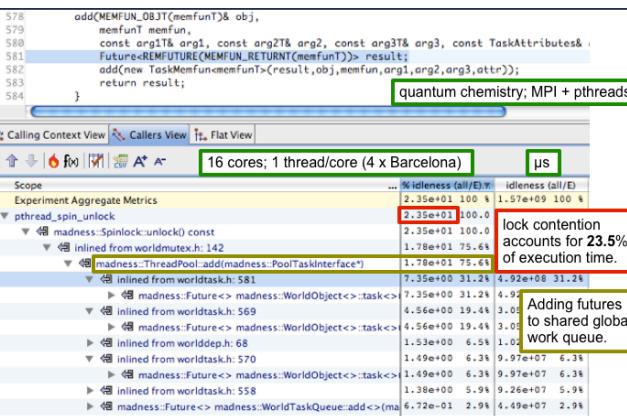
HPCToolkit Capabilities at a Glance



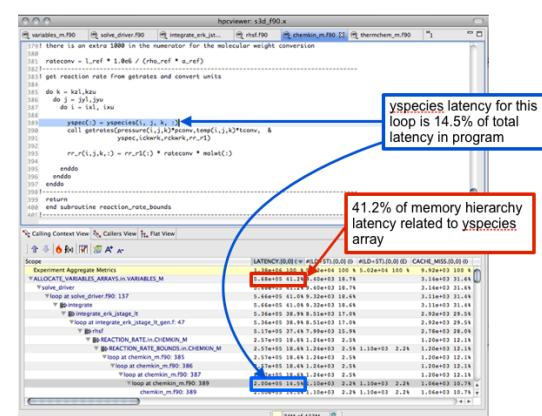
Assess Imbalance and Variability



Pinpoint & Quantify Scaling Bottlenecks



Shift Blame from Symptoms to Causes



Associate Costs with Data

hpctoolkit.org